OpenPredict - An Open Research Dataset and Evaluation Protocol for Fine-grained Predictive Testing

David Brodmann[1], Erik Rodner[2]

Abstract: Systematic testing of every single component and interface is undoubtedly an important measure to handle the complex nature of current software systems. However, this comes with often neglected computational costs. The aim of this paper is therefore to cut time and resource needs by predictive testing, i.e., predicting test failures with machine learning using a surprisingly simple statistical feature representation. Furthermore, we present the first open research benchmark for predictive testing to enable and foster future research in this area.

Keywords: machine learning; software testing; research dataset; predictive testing

1 Introduction

Software testing is crucial for the success of a professional software project. However, with a growing code complexity, regression testing for changes gets more time and resource consuming resulting in hours of software tests for evaluating code changes for every new commit. A time delay, which is simply impractical to handle during development and a waste of expensive resources. A classical solution to the problem is to track test dependencies and code coverage of each test over time. However, this is time-consuming and sometimes impossible to achieve especially within heterogeneous repositories with several language barriers [MSPC19].

Because predictive testing is heavily project dependent, our contribution to the field of predictive testing is not a new machine learning approach, but rather a curated and open dataset to allow future comparison and evaluation of predictive testing methods.

In summary our main contributions are as follows:

- Repository containing all the databases and code: https://gitlab.com/nexxtnit/predictive_testing
- Ready-to-use database of test results and metadata on a commit basis for four open-source projects (Chap. 2).
- A new baseline method for predictive testing based on simple statistics (Chap.

[1] DResearch Fahrzeugelektronik, Prüflabor / KI, Spenerstr. 38, 10557 Berlin, Germany, david.brodmann@luckycloud.de
[2] Hochschule für Technik und Wirtschaft Berlin, Fachbereich 2 / KI-Werkstatt, Wilhelminenhofstr. 75A, 12459 Berlin, Germany, erik.rodner@htw-berlin.de

3).

- Evaluation of a baseline approach (statistical feature extraction) for predictive testing to set the bar for future work (Chap. 5.2)
- In-depth analysis of feature relevance on the new dataset to provide insights concerning major relevant statistics (Chap. 5.3).

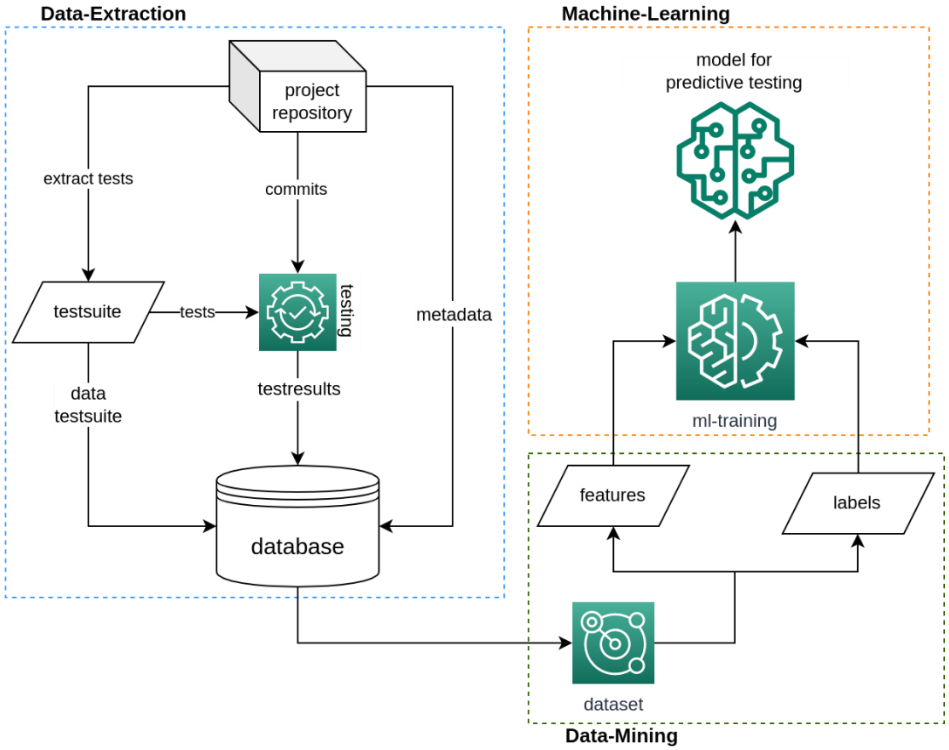

Figure 1 Illustration of predictive testing using supervised machine learning: past code changes and corresponding test results are used for training a machine learning model able to predict test failure probabilities for future code changes.

2 OpenPredict - A New Dataset for Predictive Testing

The problem of current papers and implementations of predictive testing is that the data is often very project specific and not granular enough to make general assumptions. Because predictive testing is often tuned to specific projects, publishing only the machine learning model has no benefit for further research. Furthermore, releasing predictive testing training data of closed-source implementations could reveal insights about the software

product. The data used from continuous integration pipelines is often taken as is, there is no possibility in tuning the scope of tests or choose the granularity of test execution.

In contrast, the goal of this work is to create an open research dataset for experiments in the field of predictive testing that can be systematically compared also in future work. Our dataset tries to illustrate the minimal requirements of data that needs to be available to implement predictive testing for your own project. The dataset itself was generated from open-source projects and could be theoretically extended further with the data extraction pipeline we build.

Our data is comprised of four curated open-source repositories of different nature. To ease evaluation and allow stable statistical evaluation, we used the following criteria to select repositories:

- Written in Python to allow for applying language-specific features in the future
- Minimum of 400 commits to get enough data points over time
- Minimum of 10 tests to allow for multi-task prediction
- Testing happens with pytest or tox to ease the automatic data extraction

These requirements, yielded in four repositories outlined in Table 1. In the following, we explain the process of data extraction and dataset curation in detail.

Project	Description	Commits	Testing	Link
Flit	” Simplified packaging of Python modules” [Pypa21]	1017	Tox with Pytest	Github
Mock	” The Python mock library” [Test21]	1277	Pytest	Github
Flake8	”flake8 is a python tool that glues together pycodestyle, pyflakes, mccabe, and thirdparty plugins to check the style and quality of some python code.” [Pycq21]	2074	Tox with Pytest	Github
Nox	” Flexible test automation for Python” [Thea22]	439	Pytest	Github

Table 1 Repositories used in our dataset for predictive testing.

Data Extraction for Test Change Representation: Our method for data extraction is illustrated in Figure 2. The approach is identical for all four projects being tested in this work. All information is gained from the project repository in an automatic fashion. The test suite is extracted from the existing tests and test variations of the repository. We do not generate synthetic code defects as in [Lund19], to obtain more test results. The commit hash provides us with a unique identifier for the test result as well as the related code

changes.

In contrast to [PaPr21],our predictive testing paradigm is to forecast the test result of every commit. *Every commit in the repository gets tested by every test in the test suite. As a result of that approach, we gain precise data on code changes and test results.*

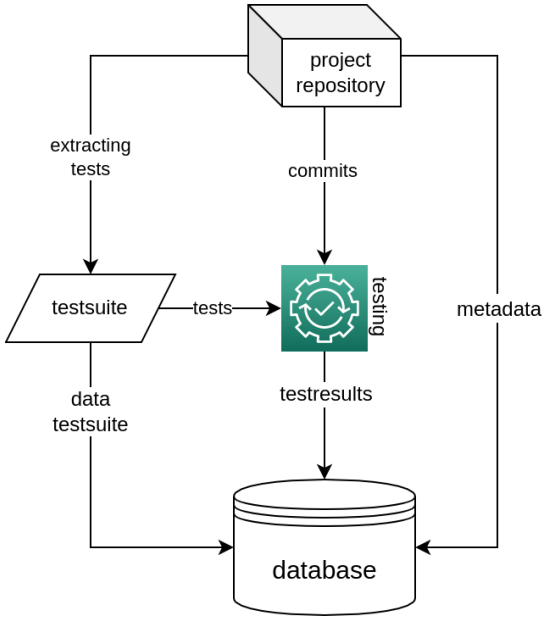

Figure 2 Data extraction method, every commit is tested with all test variations. The data gathered from the test run and the repository is saved in an SQL database keeping the correspondence between them.

Database: All meta information and main statistical features are stored in an SQL database, which is outlined in Figure 3 and identical for all projects. The scripts used for automatically transforming a git repository to the database are available in our associated code repository.

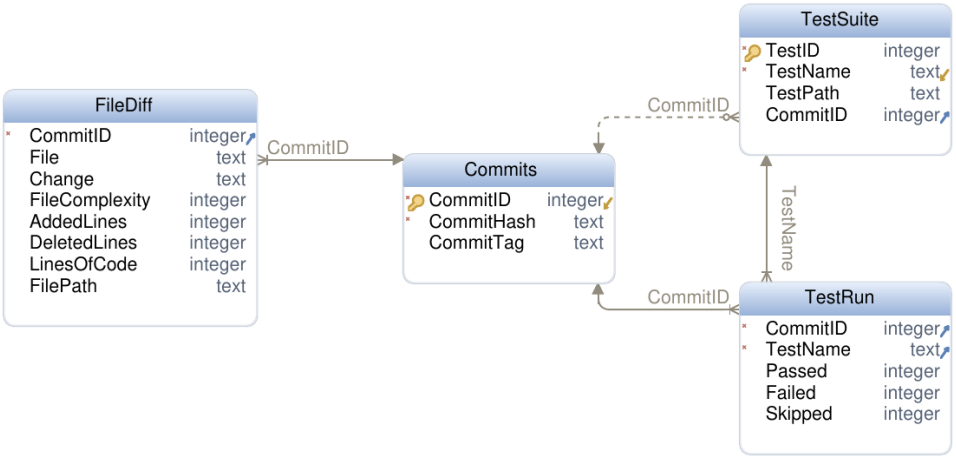

Figure 3 Database architecture, the commit-id, based on the commit hash, is used for synchronization throughout the database

Test Suite: The goal of our predictive testing approach is to predict the test result $y^{(j)} \epsilon \{\text{success, fail}\}$ of all tests $T^{(j)}$ in the test suite. However, tests can change over time. and these variations require special care. We need to add each test variation k of a test $T^{(j)}$ as an additional target $y^{(j,k)}$ to prevent the model from learning wrong statistical dependencies of older code changes and test results not related to the modified tests. We therefore include all changes of a test and create a separate test target for each change (see Table 2). All tasks $y^{(j,k)}$ for a fixed j will be not statistically independent, since test changes will consist of tiny changes over time. This fact can be exploited by a multi-task approach for learning.

With including every change in a test case as an own test variation, we increased the amount of test cases from 12 tests to 329 - phrasing each of these variations as independent prediction tasks.

Tests Newest Commit	**Test Variations in Test Suite**
test_parametrize.py	test_parametrize_354.py
	test_parametrize_359.py
	test_parametrize_367.py
test_version.py	test_version_348.py
	test_version_367.py
test_command.py	test_command_4.py
	test_command_4.py
	test_command_9.py
	test_command_24.py

...	…

Table 2 Left: tests from the newest commit of Flit; Right: test variations

For the flit project the dataset statistics can be checked in Table 3.

Version (Commits)	All (Passed/Failed)	Training Set (P/F)	Testing Set (P/F)
Flit.V0.0 (634)	208’586 (33’391/175’195)	183’253 (28’472/154’781)	25’333 (4919/20’414)
Flit.V1.0.WV (538)	177’002 (28’764/148’238)	157’920 (28’764/143’611)	19’082 (3812/15’270)
Flit.V1.0 (538)	10’760 (7591/3169)	9600 (6830/2770)	1160 (761/399)

Table 3 Dataset statistics of the Flit dataset variations. The newest 10% of the commits are for the testing set, the other 90% go in the training set, see Chapter 5.1 for details.

3 Predictive Testing with Statistical Features

In the following, we outline our baseline machine learning approach for predictive testing, which will be evaluated in Sect. 5.

A dataset consists of three parts, the one-hot encoded test results, the one-hot encoded file changes, and metadata per commit. As label data the pass/fail column is used. A snippet of the complete dataset can be examined in Table 4*.*

CommitID	test_build_422.py	test_build_465.py	…	pyproject.toml
1028	1	0	...	1
1028	0	1	…	1
...	...	...	...	…

FileComplexity	AddedLines	DeletedLines	PassFail
27	5	5	0
27	5	5	0
…	…	…	...

Table 4 Complete dataset from the Flit database for machine learning. All file changes and tests are one-hot encoded. The PassFail column is the label we try to predict, and the CommitID will be removed for training and testing.

As a classification model we use a random forest classifier to predict the outcome of a testcase as pass or fail. The random forest classifier consists of several decision trees trained with a random fraction of the training data. A decision tree is made up of inner

nodes (split nodes) and leaves. At each split node a single feature is evaluated since we make use of simple decision stumps. After traversing a tree, the final estimate is given by an empirical probability stored at each leaf during learning. A random forest decision is determined by the average of all trees estimates and a majority vote is used for the final classification decision.

Due to the simple decision stumps used, a random forest classifier can easily cope with features of different magnitudes. Furthermore, we also exploit multi-task learning of all different target tasks (every test variation is a single target) since the tasks are highly related. This is done in our case by a one-hot-encoding of the task itself and a single binary variable as a target. Some branches in a single decision tree can therefore focus on an arbitrary set of tasks by using their one-hot encodings as features in early split nodes. This is beneficial compared to learning each predictive testing task independently from each other with only a few training examples available (each test variation might only relate to a few commits).

For our experiments, we use a random forest with 1000 decision trees and a Gini split criterion without a restriction of a maximum number of examples in a leaf.

4 Related Work

Reducing the number of executed tests in regression test suites is an established method for reducing costs and time [YoHa12]. Test selection is a very efficient method for reducing the executed tests [LHSL16]. Combining test selection with machine learning is the usually referred to as predictive Testing. Despite its potential, relevance for sustainable development, and partial adoption at large tech companies [RiMS21], predictive testing is a rather unexplored area of research. In [MSPC19], Machalica et al. also used historical data to implement a data-driven test selection strategy. This allowed the authors to reduce the total number of test executions by 2/3 while still finding 95% of all failed tests. Unfortunately, no open dataset was provided, rendering reproduction of the results impossible. [Lund19] implemented a predictive test selection tool by creating synthetic test results. They altered the code with small synthetic modifications to obtain labeled data with a decent number of test failures. Classifying tests results was done using a random forest classifier, which allows for reducing test executions by 50% at again 95% true positive detection rate for all failures. A severe disadvantage of this approach is that its performance highly depends on the realistic nature of the synthetic code modifications. In complex software systems, these are difficult to design and might be in addition subject to change over time. In contrast, the approach of [SKPS20] uses simple text similarity metrics to rank tests related to a code change. In comparison to the impressive dataset gathered in [YBKB22] from 25 open-source projects, we concentrate on the basic needs to implement predictive testing. In their work, Java projects are exclusively used along with their continuous integration history. As a result, many additional data features like code coverage are available. Furthermore, using only continuous integration data has the

disadvantage that the granularity of code changes cannot be chosen by oneself. The code changes do differ from build to build and can range from one to several commits. In our work, we concentrated on changes and test results per single commit - providing predictive testing at a fine-grained level. However, the three “high level features” having the most impact on prediction in [YBKB22]are like the ones used in our work. The work of Sharif et al. [ShML21] focuses on deep learning techniques for regression test prediction. They show the benefits of their approach especially for large datasets.

A related problem to ours is test suite failure prediction [PaPr21],which tries to predict the failure of the whole test suite rather than for each test case individually. A further overview of predictive methods for software engineering is given in [YXLB22].

5 Experiments

In the following, we evaluate our baseline approach on OpenPredict showing the power of simple statistical features for predictive testing.

5.1 Experimental Setup

For experimenting with machine learning methods for test case prediction, the dataset needs to be separated first in training and testing data. Instead of selecting random cases from the dataset, we decided to split the data by its commit-id, i.e., trying to predict the behaviour of newer commit from the history of older ones. This reflects the use case for predictive testing, where the model is trained continuously and applied to the most recent commit. While analyzing the databases, it became obvious that some commits, did not have any test results. This is because early commits in the repository have often not been able to execute tests at all, therefore they contain no test results.

The dataset is split into testing and training data by the CommitID. From the sum of commits containing test data, the newest 10% are used for testing, the other 90% for training. Please note that the commit-id is of course not used as a feature.

Since our prediction tasks are binary, we used an ROC analysis as the main performance metric for our experiments. Instead of calculating ROC results for each task, we evaluated the performance of the random forest classifier, which provides us with an aggregated performance metric over all test cases.

Dataset	Specifications	Features
Flit.V0.0	• no metadata • all commits, starting at init • all test cases, no filtering of variations	• test • changed files in commit • test result

Flit.V1.0.WV	• metadata *FileComplexity* and *AddedLines* • starting at commit tag *V1.0* • all test cases, no filtering of variations	• test • changed files in commit • test result • *FileComplexity* • *AddedLines*
Flit.V1.0	• Metadata *FileComplexity* and *AddedLines* added • Starting at commit tag V1.0 • Combining all test variations to one test case	• test • changed files in commit • test result • *FileComplexity* • *AddedLines*

Table 5 Dataset variations of the Flit database: Flit.V0.0 has all raw data and no metadata, Flit.V1.0.WV has metadata added, Flit.V1.0 contains structured test results and metadata.

5.2 Analysis on the flit dataset

First, we evaluate some dataset aspect on the Flit dataset with our baseline approach. In particular, we want to evaluate the impact of skipping the very first commits and dealing with test case variations as individual tasks (see Figure 5 for an illustration). Our experiments are performed with three different versions of the Flit dataset in Table 5. The results are given in Figure 4.

The AUC improves about 6% when adding more relevant data to the dataset and about 8% when structuring the test results of the testcases according to their commit-id.

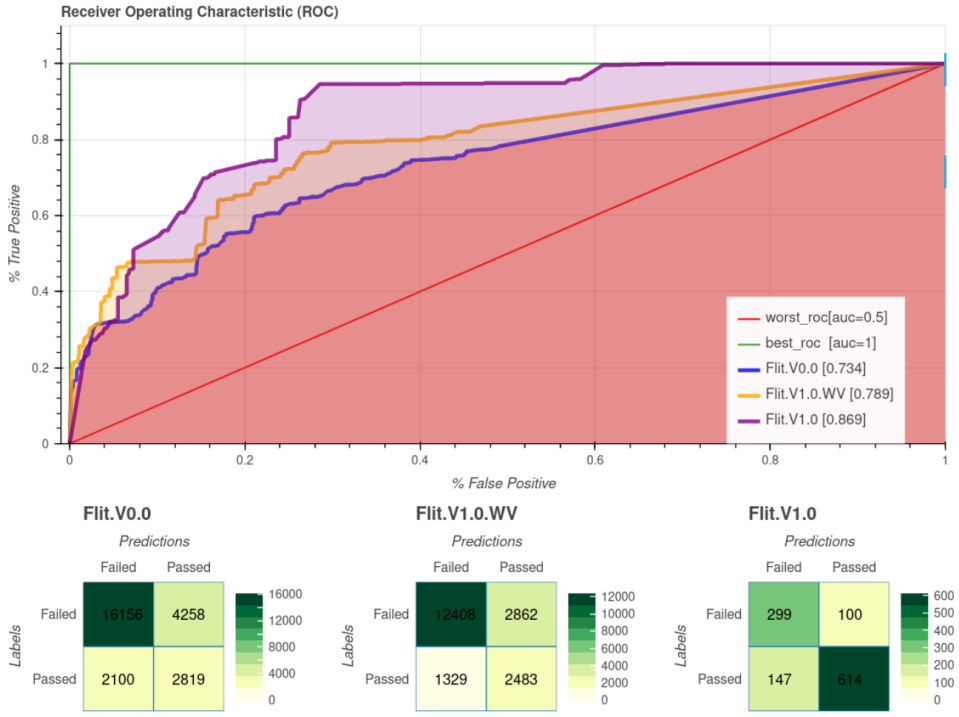

Figure 4 Different classification for the three dataset variations of Flit from Tab. 5

For improving classification an attempt is made to restructure the data of all test case variations. Because all test case variations test the same code functionality, logically it makes sense to only use the result of one test variation at a time. At every commit the result of the matching test variation must be used.

*As a result of the previous observations, a test variation is used as long as the commit-id of the current commit to test, is one greater than the commit-id of the test variation (*Figure 5*).*

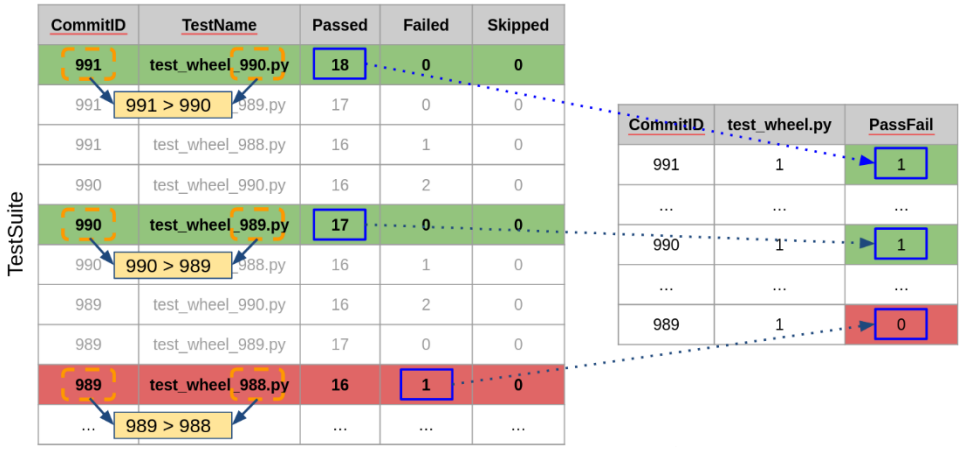

Figure 5 Test variations being restructured for a dataset. The Results of every test variation is used if the commit currently testing is greater than the commit the test variation is created.

5.3 In-depth analysis of feature relevance

The metadata features we identified in Sect. 3 are contributing strongly to the classification. The importance is visible in Table 6. The feature importance is measured as the impurity decrease within each tree. The absolute value of the accumulated impurity decrease is the importance of each feature. Besides the obvious test cases as features, the metadata features are the ones with the biggest impact.

Nr.	Importance	Feature
1	0.137	test_importable_.py
2	0.119	test_metadata_.py
3	0.098	**FileComplexity**
4	0.055	test_config_.py
5	0.052	**AddedLines**
6	0.049	**DeletedLines**

Table 6 Feature importance of Flit.V1.0 dataset. The Metadata features FileComplexity, AddedLines and DeletedLines are within the 6 most important features.

When analyzing the classification results, the question emerged what probably would happen, if the commit-id is kept as a feature in the dataset. The results can be seen in Figure 6.

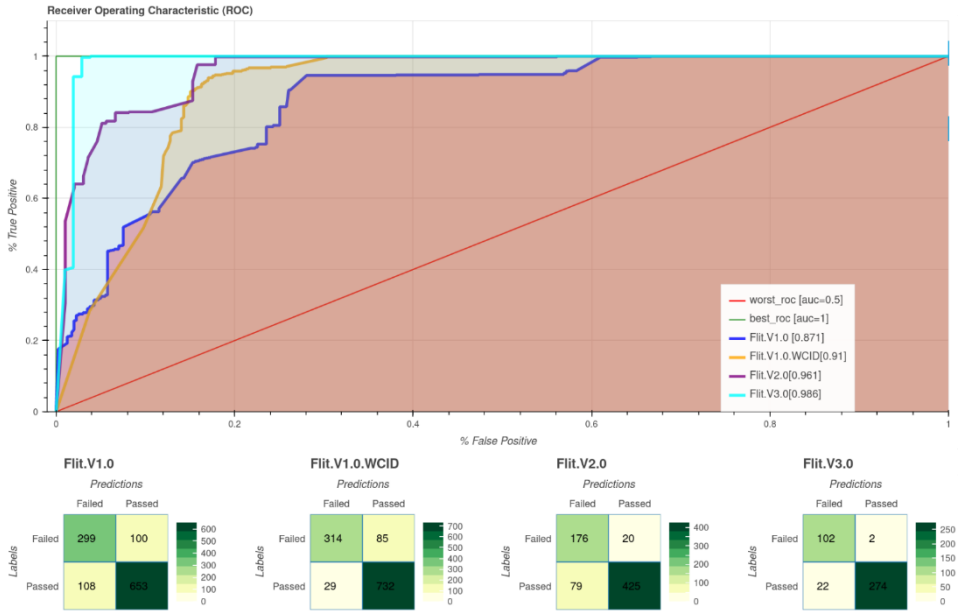

Figure 6 Importance of commits included in dataset

Flit.V1.0: The blue line in the roc-analysis (Figure 6) and the classification matrix on the left border, is the best performing dataset from Chap. 5.1 (Flit.V1.0).

Flit.V1.0.WCID: The orange line and second classification matrix from the left (Figure 6), is the same dataset as Flit.V1.0 but with the feature commit-id added.

The AUC of the random forest classifier is improving by around 5% by adding the commit-id as a feature to the dataset.

The reason for the performance improvement can be found in a deeper analysis of the nodes of the feature commit-id in the balanced random forest classifier, visible in Figure 7.

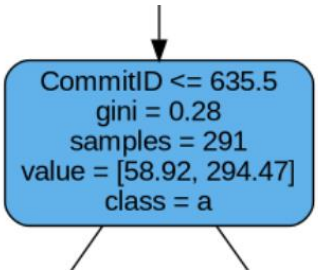

Figure 7 Node for feature commit-id in tree of balanced random forest classifier

The whole dataset contains around 1000 commits. Starting with commit-id 1 for the oldest commit, the newest commit has the id 1000. The node in Figure 7 is separating the commits for the commit-id value <= 635, 5. Because our test set contains only the newest 10% of the commits, the commit-id in the test set will always be greater than 635,5. This observation leads to the idea that fewer commits in the dataset could actually benefit the classifier.

Flit.V2.0: The purple line in the ROC analysis and the classification matrix second from the right border, is the same dataset as the first one. The commit-id is not included as a feature, but commits were only used starting with commit-tag V2.0.

Flit.V3.0: The cyan line in the ROC analysis and the classification matrix first from the right, is the same dataset as the first one. The commit-id is not included as a feature, but commits were only used starting with commit-tag V3.0.

In conclusion, it becomes clear that a logical connection of the code, represented with version tags, is important to train a machine learning model for predictive testing. This leads to the theory that not the whole history of a repository is necessary for predictive testing, but rather the last one or two versions of the project code.

Version (Commits)	**All (Passed/Failed)**	**Training Set (P/F)**	**Testing Set (P/F)**
Flit.V1.0 (538)	10’760 (7591/3169)	9600 (6830/2770)	1160 (761/399)
Flit.V1.0.WCID (538)	10’760 (7591/3169)	9600 (6830/2770)	1160 (761/399)
Flit.V2.0 (347)	6940 (4948/1992)	6240 (4444/1796)	700 (504/196)
Flit.V3.0 (193)	3860 (2678/1182)	3460 (2382/1078)	400 (296/104)

Table 7 Statistical features of the datasets used to analyze the commit-id feature

5.4 Full evaluation on OpenPredict

In Table 8 and Figure 8 the size of all datasets and the performance of the classifiers for the different projects can be examined. All four projects we tested in our work provided an acceptable classification. The visible differences in Figure 8 are explainable because of the code complexity, size, and test amount of the different projects. The worst classification project, *Flake8*, also offers space to further improve classification performance by increasing the commit tag.

Training a predictive testing model for different repositories shows that our approach is not project specific and offers the possibility to implement predictive testing in a general manner, as long as the correct data is provided.

Version (Commits)	**All (Passed/Failed)**	**Training Set (P/F)**	**Testing Set (P/F)**
Flit.V2.0 (347)	6940 (4948/1992)	6240 (4444/1796)	700 (504/196)
Mock.V0.8 (464)	4640 (2636/2004)	4170 (2254/1916)	470 (382/88)
Nox.V0.0 (436)	6976 (2964/4012)	6272 (2486/3786)	704 (478/226)
Flake8.V3.7 (613)	27’585 (10’178/17’407)	24’795 (8564/16’231)	2790 (1614/1176)

Table 8 Statistical features of the datasets of all different projects

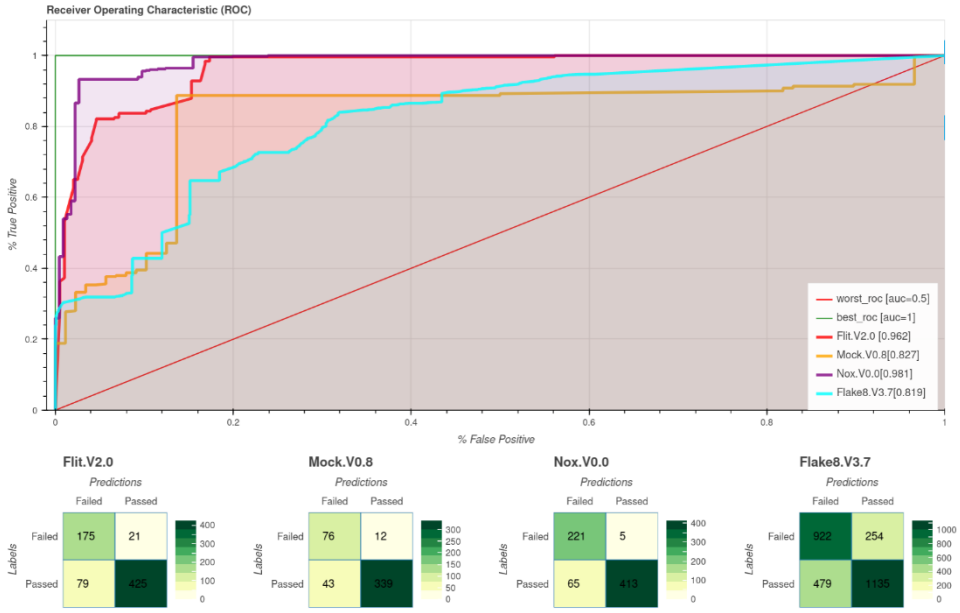

Figure 8 Overview of the best possible classification with optimizations discovered in this paper.

Conclusions and Future Work

We presented the first open research dataset for predictive testing on a fine-granular basis. With this minimalistic approach we identified the core features necessary to predict test results. With a detailed analysis of feature importance, it was even possible to conclude that not the whole code history is necessary for gathering training data. Furthermore, we developed a baseline approach based on statistical features from the repository and a multi-task random forest as predictor. Applied to the dataset, the approach showed surprisingly high prediction performance, even though no detailed code analysis was used to enrich the feature representation of the related code changes.

There is a multitude of research ideas to boost the performance of predictive testing, including using recent language models trained on source code [CTJY21] or even directly on code changes. Although we use a simple multi-task technique (using one-hot-encoded task descriptions) for prediction, this representation lacks the information that variations of the same underlying unit test are related differently based on their commit history. In addition, the baseline approach and related ones should be further evaluated in a continuous learning setting, where the model is learned from last K commits and applied to the most recent one.

Bibliography

[CTJY21] Chen, Mark ; Tworek, Jerry ; Jun, Heewoo ; Yuan, Qiming ; Pinto, Henrique Ponde de Oliveira ; Kaplan, Jared ; Edwards, Harri ; Burda, Yuri ; u. a.: Evaluating large language models trained on code. In: *arXiv preprint arXiv:2107.03374* (2021)

[LHSL16] Legunsen, Owolabi ; Hariri, Farah ; Shi, August ; Lu, Yafeng ; Zhang, Lingming ; Marinov, Darko: An extensive study of static regression test selection in modern software evolution: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016. In: Su, Z. ; Zimmermann, T. ; Cleland-Huang, J. (Hrsg.) *FSE 2016 - Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*., Association for Computing Machinery (2016), S. 583–594

[Lund19] Lundsten, Erik: *EALRTS : A predictive regression test selection tool*, KTH, School of Electrical Engineering and Computer Science (EECS), Master's Thesis, 2019. — Backup Publisher: KTH, School of Electrical Engineering and Computer Science (EECS)

[MSPC19] Machalica, Mateusz ; Samylkin, Alex ; Porth, Meredith ; Chandra, Satish: Predictive Test Selection. In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, *ICSE-SEIP '19* : IEEE Press, 2019, S. 91–100

[PaPr21] Pan, Cong ; Pradel, Michael: Continuous test suite failure prediction. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, S. 553–565

[Pycq21] Pycqa: flake8. *GitHub*.

[Pypa21] pypa: flit. *GitHub*.

[RiMS21] Ricca, Filippo ; Marchetto, Alessandro ; Stocco, Andrea: AI-based Test Automation: A Grey Literature Analysis. In: *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* : IEEE, 2021, S. 263–270

[ShML21] Sharif, Aizaz ; Marijan, Dusica ; Liaaen, Marius: DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* : IEEE, 2021, S. 525–534

[SKPS20] Sutar, Shantanu ; Kumar, Rajesh ; Pai, Sriram ; Shwetha, BR: Regression test cases selection using natural language processing. In: *2020 International Conference on Intelligent Engineering and Management (ICIEM)* : IEEE, 2020, S. 301–305

[Test21] testing-cabal: mock. *GitHub*.

[Thea22] theacodes: nox. *GitHub*.

[YBKB22] Yaraghi, Ahmadreza Saboor ; Bagherzadeh, Mojtaba ; Kahani, Nafiseh ; Briand, Lionel: Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. In: *IEEE Transactions on Software Engineering* (2022), S. 1–24

[YoHa12] Yoo, S. ; Harman, M.: Regression testing minimization, selection and prioritization: a survey. In: *Software Testing, Verification and Reliability* Bd. 22 (2012), Nr. 2, S. 67–120

[YXLB22] Yang, Yanming ; Xia, Xin ; Lo, David ; Bi, Tingting ; Grundy, John ; Yang, Xiaohu: Predictive Models in Software Engineering: Challenges and Opportunities. In: *ACM Transactions on Software Engineering and Methodology* Bd. 31 (2022), Nr. 3, S. 56:1-56:72